CS 61C:
Great Ideas in Computer Architecture
Intro to Assembly Language, MIPS Intro

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., MIPS)

Anything can be represented as a *number*, i.e., data or instructions

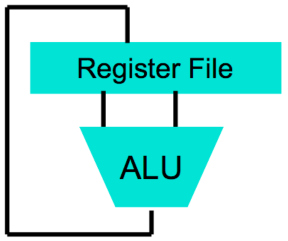
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

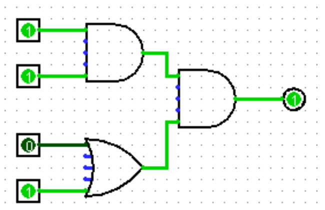
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



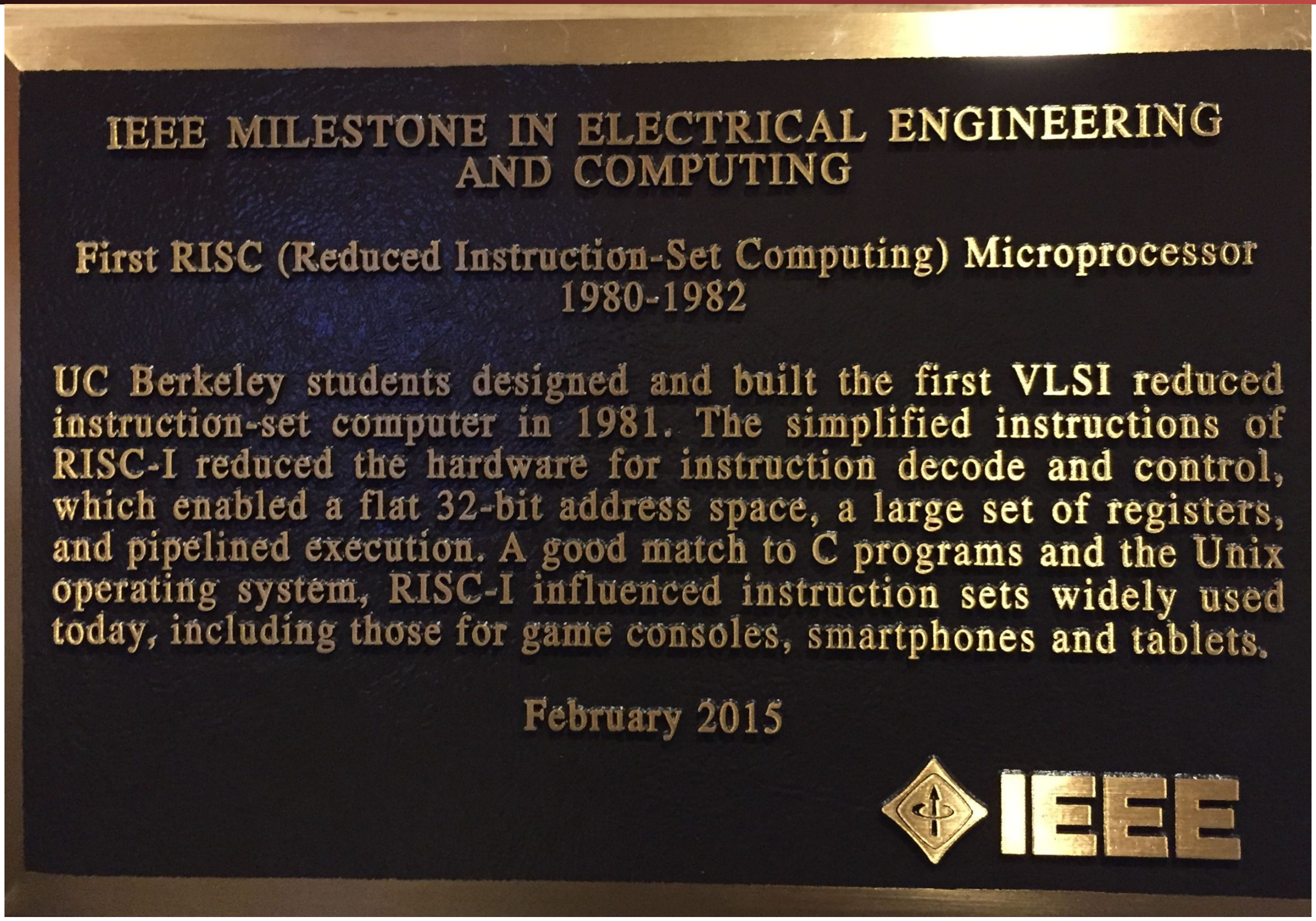
Assembly Language

- Basic job of a CPU: execute lots of ***instructions***.
- Instructions are the primitive operations that the CPU may execute
  - The stored program in "stored program computer"
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an Instruction Set Architecture (ISA).
  - Examples: ARM, Intel x86, MIPS, RISC-V, IBM/Motorola PowerPC (old Mac), Motorola 68k (really old Mac), Intel IA64 (aka Itanic), ...

Instruction Set Architectures

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
  - VAX architecture had an instruction to multiply polynomials!
- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) Reduced Instruction Set Computing
  - Keep the instruction set small and simple, makes it easier to build fast hardware
  - Let software do complicated operations by composing simpler ones

Berkeley Acknowledge for the first RISC computer



IEEE MILESTONE IN ELECTRICAL ENGINEERING
AND COMPUTING

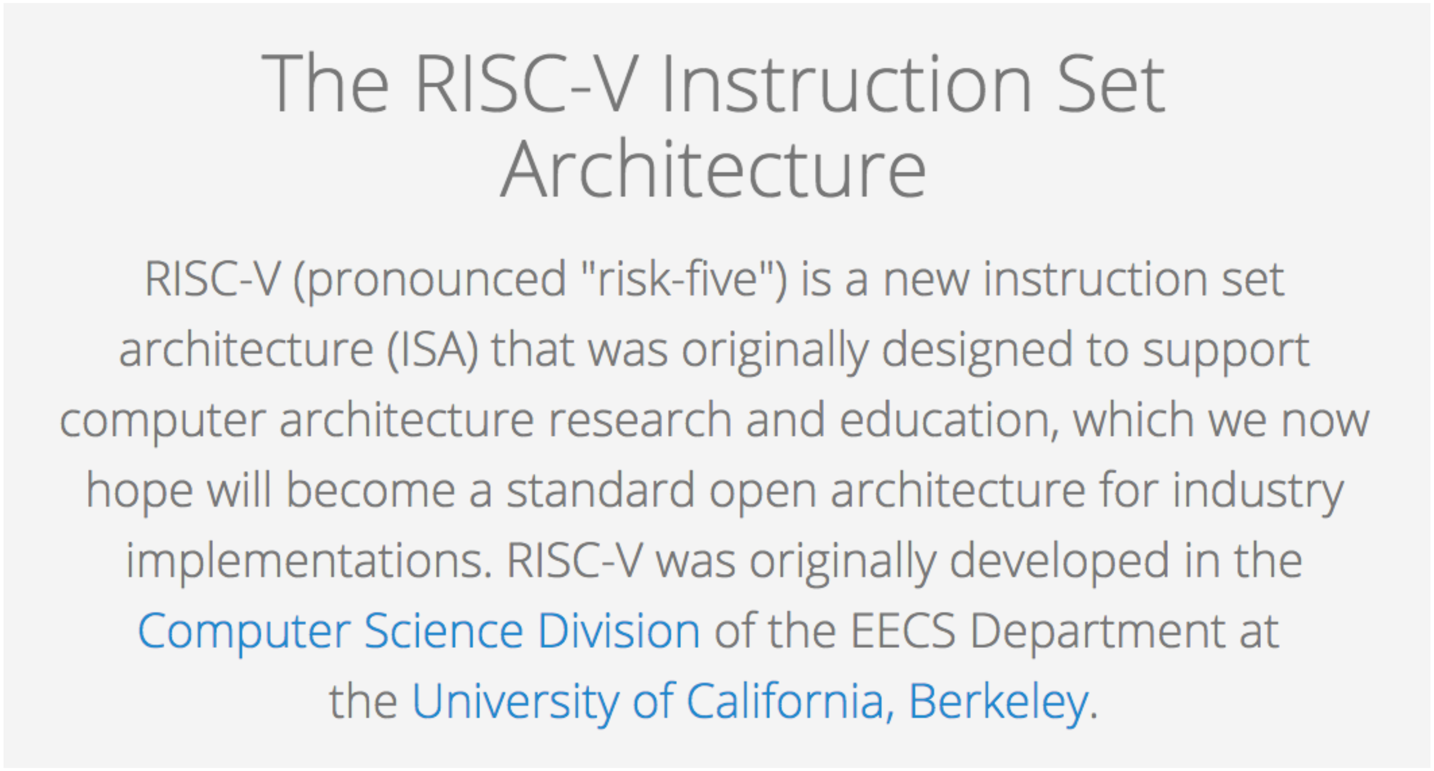
First RISC (Reduced Instruction-Set Computing) Microprocessor
1980-1982

UC Berkeley students designed and built the first VLSI reduced instruction-set computer in 1981. The simplified instructions of RISC-I reduced the hardware for instruction decode and control, which enabled a flat 32-bit address space, a large set of registers, and pipelined execution. A good match to C programs and the Unix operating system, RISC-I influenced instruction sets widely used today, including those for game consoles, smartphones and tablets.

February 2015

IEEE

From RISC-I to RISC-V



The RISC-V Instruction Set Architecture

RISC-V (pronounced "risk-five") is a new instruction set architecture (ISA) that was originally designed to support computer architecture research and education, which we now hope will become a standard open architecture for industry implementations. RISC-V was originally developed in the Computer Science Division of the EECS Department at the University of California, Berkeley.

EECS151/251A, CS152, CS250, CS252 use RISC-V

But at the same time... All RISCs are mostly the same except for one or two silly design decisions

- MIPS:
  - "The Branch Delay Slot": The instruction immediately after an if is always executed
- SPARC:
  - "Register Windows"
- ARM:
  - 32b is ugly as sin...
  - But 64b is pretty much generic RISC with condition codes for instructions: Allow you to say "only do this instruction if a previous value was TRUE/FALSE"

MIPS Architecture

- MIPS – semiconductor company that built one of the first commercial RISC architectures
- We will study the MIPS architecture in some detail in this class
  - When an upper division course uses RISC-V instead, the ISA is very similar)
- Why MIPS instead of Intel x86?
  - MIPS is simple, elegant. Don’t want to get bogged down in gritty details.
  - MIPS widely used in embedded apps, x86 little used in embedded, and more embeded computers than PCs
    - And you are more likely to need C and/or assembly in dealing with embedded systems
- Why MIPS instead of ARM?
  - Accident of history: Patterson & Hennesey is in MIPS
  - MIPS also designed more for performance than ARM, ARM is instead designed for small code size

Assembly Variables: Registers

- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly Operands are ***registers***
  - Limited number of special locations built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast
  (faster than 1 ns - light travels 30cm in 1 ns!!!)

Number of MIPS Registers

- Drawback: Since registers are in hardware, there are a predetermined number of them
  - Solution: MIPS code must be very carefully put together to efficiently use registers
- 32 registers in MIPS
  - Why 32? Smaller is faster, but too small is bad. Goldilocks problem.
    - Plus can use 5 bits to specify the register
  - X86 has many fewer registers
    - Under the hood, a modern x86 actually translates to an internal RISC!
- Each MIPS register is 32 bits wide
  - Groups of 32 bits called a word in MIPS

Names of MIPS Registers

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:
  - $0, $1, $2, … $30, $31
- For now:
  - $16 - $23➔ $s0 - $s7 (correspond to C variables)
  - $8 - $15 ➔ $t0 - $t7 (correspond to temporary variables)
  - Later will explain other 16 register names
- In general, use register names to make your code more readable

C, Java variables vs. registers

- In C (and most High Level Languages) variables declared first and given a type
  - Example: **`int fahr, celsius; char a, b, c, d, e;`**
- Each variable can ONLY represent a value of the type it was declared
  - Can't treat an **`int`** like a **`char`**, and converting (casting) a **`char`** to an **`int`** changes type.
- In Assembly, registers have no type:
  - The operation determines how to treat the register's contents

Addition and Subtraction of Integers

- Addition in Assembly
  - MIPS: `add $s0,$s1,$s2`
    C: `a = b + c`
  - where C variables ⇔ MIPS registers are:

    `a ⇔ $s0, b ⇔ $s1, c ⇔ $s2`

- Subtraction in Assembly
  - MIPS: `sub $s0,$s1,$s2`
    C: `a = b - c`
  - where C variables ⇔ MIPS registers are:

    `a ⇔ $s0, b ⇔ $s1, c ⇔ $s2`

Addition and Subtraction of Integers Example 1

- How to do the following C statement?
  `a = b + c + d - e;`
- Break into multiple instructions

  -
    ```
    add $t0, $s1, $s2 # temp = b + c
    add $t0, $t0, $s3 # temp = temp + d
    sub $s0, $t0, $s4 # a = temp - e
    ```
- A single line of C may break up into several lines of MIPS.
- Notice the use of temporary registers – don’t want to modify the variable registers $s
- Everything after the hash mark on each line is ignored (comments)

Immediates

- Immediates are numerical constants
  - They appear often in code, so there are special instructions for them
- Add Immediate:
  - **`addi $s0,$s1,-10`** (in MIPS)
  - **`f = g - 10`** (in C)
    - where MIPS registers **`$s0,$s1`** are associated with C variables **`f, g`**
- Syntax similar to add instruction, except that last argument is a number instead of a register

Overflow in Arithmetic

- Reminder: Overflow occurs when there is a “mistake” in arithmetic due to the limited precision in computers.
- Example (4-bit unsigned numbers):

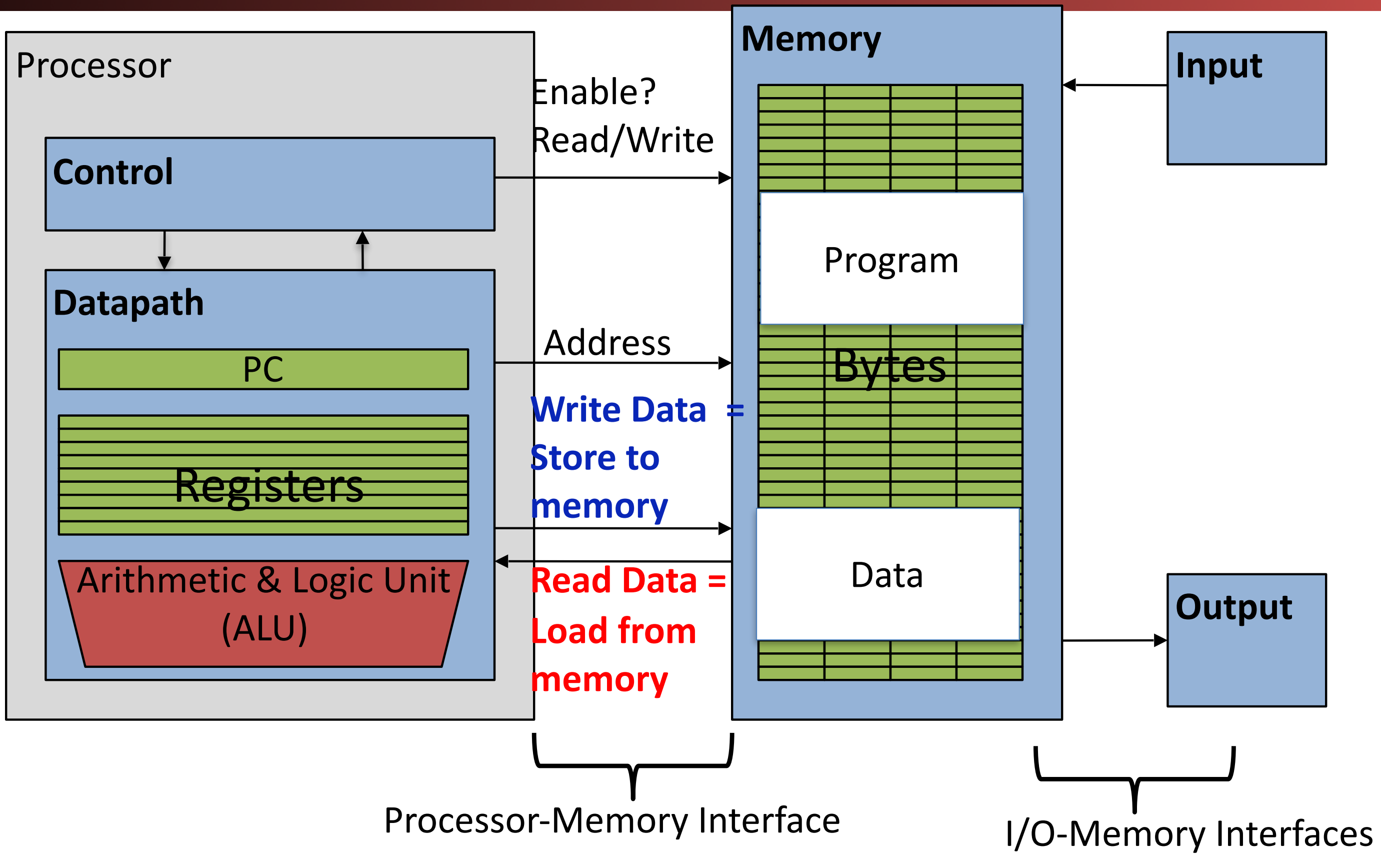
```
     15              1111
   + 3             + 0011
     18             10010
```

- But we don’t have room for 5-bit solution, so the solution would be 0010, which is +2, and “wrong”.

Overflow handling in MIPS

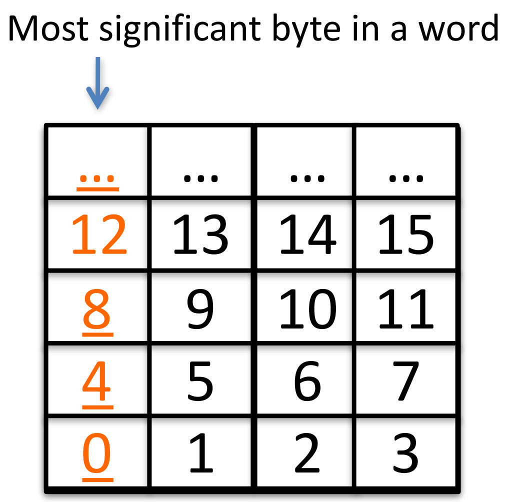
- Some languages detect overflow (Ada), some don't (C, Java)
- MIPS solution is 2 kinds of instructions
  - The ***only*** difference is what happens on overflow!
- "signed" cause overflow to be detected
  - add (**`add`**)
  - add immediate (**`addi`**)
  - subtract (**`sub`**)
- "unsigned" do not cause overflow detection
  - add unsigned (**`addu`**)
  - add immediate unsigned (**`addiu`**)
  - subtract unsigned (**`subu`**)
- Compiler selects appropriate arithmetic
  - MIPS C compilers produce **`addu, addiu, subu`**

Data Transfer: Load from and Store to memory



Memory Addresses are in Bytes

- Lots of data is smaller than 32 bits, but rarely smaller than 8 bits – works fine if everything is a multiple of 8 bits
- 8 bit chunk is called a byte
  - (1 MIPS word = 4 bytes, a "32b system")
- Memory addresses are really in bytes, not words
- Word addresses are 4 bytes apart
  - Word address is same as address of
  - leftmost byte – most significant byte (i.e. Big-endian convention)



Transfer from Memory to Register

- C code:

  -
    ```
    int  A[100];
    g = h + A[3];
    ```
- Using Load Word (lw) in MIPS:

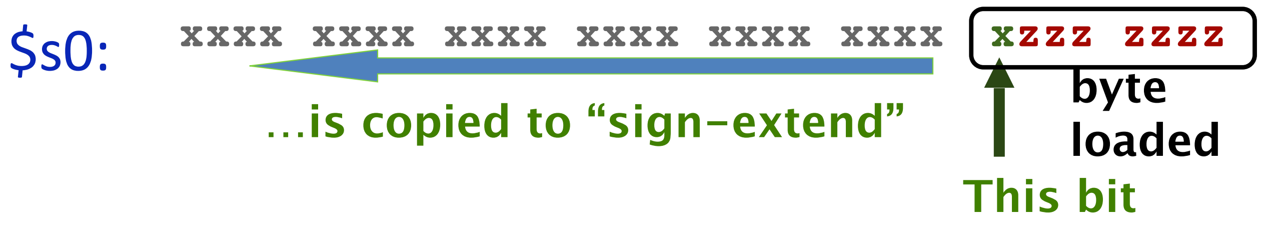
  -
    ```
    lw $t0,12($s3)  # Temp reg $t0 gets A[3]
    add $s1,$s2,$t0 # g = h + A[3]
    ```
- Note: $s3 == A – base register (pointer)
- 12 == offset in bytes
  - Offset must be a constant known at assembly time

Transfer from Register to Memory

- C code
  - `int  A[100];`
    `A[10] = h + A[3];`
- Using Store Word (sw) in MIPS:
  - **`lw  $t0,12($s3)   # Temp reg $t0 gets A[3]`**
  - **`add $t0,$s2,$t0   # Temp reg $t0 gets h + A[3]`**
  - **`sw   $t0, 40($s3) # A[10] = h + A[3]`**
- $s3+12 and $s3+40 must be multiples of 4: Word alignment!

Loading and Storing bytes

- In addition to word data transfers (**`lw, sw`**), MIPS has byte data transfers:
  - load byte: **`lb`**
  - store byte: **`sb`**
- Same format as **`lw`**, **`sw`**
  - But offsets can be individual addresses, they don't have to be word aligned
- E.g., **`lb $s0, 3($s1)`**
  - contents of memory location with address = sum of "3" + contents of register $s1 is copied to the low byte position of register $s0.



How Much Progress on the Project?

A: What's a project?

B: Seen Piazza Post

C: Gotten Code, started

D: Done with rebugging (aka "Programming")

E: Done with debugging

Clickers/Peer Instruction

We want to translate `*x = *y +1` into MIPS
(x, y pointers stored in: `$s0 $s1`)

```
A:  addi $s0,$s1,1

B:  lw   $s0,1($s1)
    sw   $s1,0($s0)

C:  lw   $t0,0($s1)
    addi $t0,$t0,1
    sw   $t0,0($s0)


D:  sw   $t0,0($s1)
    addi $t0,$t0,1
    lw   $t0,0($s0)

E:  lw   $s0,1($t0)
    sw   $s1,0($t0)
```

Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory: Billions of bytes (2 GB to 8 GB on laptop)
- and the RISC principle is…
  - Smaller is faster
- How much faster are registers than memory??
- About 100-500 times faster!
  - in terms of latency of one access
  - We will later learn about caches to reduce this latency in practice

MIPS Logical Instructions

- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called *logical operations*

Logical operations	C operators	Java operators	MIPS instructions
Bit-by-bit AND	&	&	**`and`**
Bit-by-bit OR	|	|	**`or`**
Bit-by-bit NOT	~	~	**`not`**
Shift left	<<	<<	**`sll`**
Shift right	>>	>>>	**`srl`**

Logic Shifting

- Shift Left:
  `sll $s1,$s2,2 #s1=s2<<2`
  - Store in $s1 the value from $s2 shifted 2 bits to the left (they fall off end), inserting 0's on right; << in C.
- Before:
  `0000 0002hex`
  `0000 0000 0000 0000 0000 0000 0000 0010two`
- After:
  `0000 0008hex`
  `0000 0000 0000 0000 0000 0000 0000 1000two`
- What arithmetic effect does shift left have?
- Shift Right: **`srl`** is opposite shift; >>

Arithmetic Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register `$s0` contained
  - **`1111 1111 1111 1111 1111 1111 1110 0111two= -25ten`**
- If executed **`sra $s0, $s0, 4`**, result is:
  - **`1111 1111 1111 1111 1111 1111 1111 1110two= -2ten`**
- Unfortunately, this is NOT same as dividing by 2n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

And In Conclusion …

- Computer words and vocabulary are called instructions and instruction set respectively
- MIPS is example RISC instruction set in this class
- Rigid format: 1 operation, 2 source operands, 1 destination
  - **`add,sub,mul,div,and,or,sll,srl,sra`**
  - **`lw,sw,lb,sb`** to move data to/from registers from/to memory.
  - On monday we will get control flow:
    - **`beq, bne, j, slt, slti`**
- Simple mappings from arithmetic expressions, array access, if-then-else in C to MIPS instructions